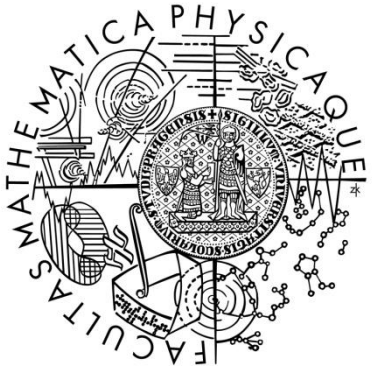
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Kristina Bártová

Hypertextury v JaGrLib
Hypertextures in JaGrLib

Department of Software and Computer Science Education

Thesis supervisor: RNDr. Josef Pelikán
Study program: Computer science, General computer science

Prague 2011

I would like to thank my thesis supervisor RNDr. Josef Pelikán for many valuable suggestions and for all the time spent at consultations. Further, I would like to thank my parents and family for their support. My big gratitude also belongs to Martina Štáflová, her husband and my brother for language correcting.

I hereby certify that I wrote the thesis myself using only the referenced sources. I agree with publishing and lending of this thesis.

In Prague, 16th May 2011 Kristina Bártová

Contents

1 Ray tracing 8

1.1. Introduction 8

1.2. Algorithm overview 9

1.3. Constructive solid geometry 11

1.4. Ray tracing recursive implementation 11

1.5. Limitations of ray tracing 12

2 Hypertextures 13

2.1. Introduction 13

2.2. Hypertextures 14

2.3. Density Modulation Functions 15

2.4. Ray marching overview 16

2.5. JaGrLib 18

2.6. Ray tracing in JaGrLib 19

3 Ray marching implementation 21

3.1. Introduction 21

3.2. Ray marching 21

3.3. Soft objects 22

3.4. DMFs 23

3.5. Scenes 23

4 Results and discussion 25

4.1. Time complexity 37

5 User guide 38

5.1. JaGrLib and Skel 38

5.2. Starting application 38

5.3. Changing the scene 38

5.4. Changing properties of soft objects 39

5.5. Result 41

6 Conclusion 42

Bibliography 43

Appendix 45

List of figures

Figure 1.1: Photograph or computer graphics? 8
Figure 1.2: Ray tracing 9
Figure 1.3: Real world 10
Figure 1.4: Backward ray tracing 10
Figure 1.5: Boolean union 11
Figure 1.6: Boolean difference 11
Figure 1.7: Boolean intersection 11
Figure 2.1: Texture mapping in two dimensions 13
Figure 2.2: Texture mapping in three dimensions 13
Figure 2.3: Sphere with a soft region 14
Figure 2.4: Simplified model of ray marching process 17
Figure 2.5: Composition of ray tracing 19
Figure 4.1: Sphere with a soft region 25
Figure 4.2: Cube with a soft region 26
Figure 4.3: Influence of march size (= 1,0) 26
Figure 4.4: Influence of march size (= 0,8) 27
Figure 4.5: Influence of march size (= 0,5) 27
Figure 4.6: Influence of march size (= 0,01) 27
Figure 4.7: Influence of frequency (= 1) 28
Figure 4.8: Influence of frequency (= 5) 28
Figure 4.9: Influence of frequency (= 12) 28
Figure 4.10: Influence of frequency (= 25) 29
Figure 4.11: Influence of frequency (= 50) 29
Figure 4.12: Frequency = 40, amplitude = 0,25 30
Figure 4.13: Frequency = 20, amplitude = 0,5 30
Figure 4.14: Frequency = 5, amplitude = 2,0 30
Figure 4.15: Frequency = 2, amplitude = 5,0 31
Figure 4.16: Frequency = 1, amplitude = 10,0 31

Figure 4.17: Softness influence 32
Figure 4.18: Softness influence 32
Figure 4.19: Softness influence 32
Figure 4.20: Number of steps (= 1) 33
Figure 4.21: Number of steps (= 2) 33
Figure 4.22: Number of steps (= 3) 33
Figure 4.23: Number of steps (= 5) 34
Figure 4.24: Number of steps (= 20) 34
Figure 4.25: FractalHypertexture frequency = 1 35
Figure 4.26: FractalHypertexture frequency = 3 35
Figure 4.27: FractalHypertexture frequency = 5 35
Figure 4.28: FractalHypertexture frequency = 15 36
Figure 4.29: Soft object and solid objects 36
Figure 5.1: Scene module in Skel 39
Figure 5.2: Different thickness of the soft region of a cube 40
Figure 5.3: Bit-streams module 41

Název práce: Hypertextury v JaGrLib
Autor: Kristina Bártová
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
E-mail vedoucího: Josef.Pelikan@mff.cuni.cz

Abstrakt: Cílem této práce je implementovat a otestovat algoritmus ray marching v grafické knihovně JaGrLib. Pomocí tohoto algoritmu budeme moci vykreslovat tzv. hypertextury – objekty s neuvěřitelně komplexním povrchem či naopak s povrchem, který není přesně definovatelný. V práci budeme navazovat na již implementovaný algoritmus ray tracing, který rozšíříme o metodu march. Ray marching otestujeme na tzv. soft objektech a funkce aplikované k vytvoření hypertextur budeme definovat pomocí šumových a fraktálových funkcí.

Klíčová slova: ray marching, metoda sledování paprsku, hypertextura, soft objekt, fraktál, šumová funkce, počítačová grafika, JaGrLib, Skel

Title: Hypertextures in JaGrLib
Author: Kristina Bártová
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract: The object of this work is to implement and test ray marching algorithm in graphical library JaGrLib. Through the use of this algorithm, we will be able to render hypertextures – objects with very complex surface or with not well defined boundary surface. In this project we will make use of already implemented ray tracing algorithm, which we will extend to the march method. Ray marching will be tested on soft objects and the functions used to create hypertextures will be defined by noise and fractal functions.

Keywords: ray marching, ray tracing, hypertexture, soft object, fractal, noise function, computer graphics, JaGrLib, Skel

Chapter 1

Ray tracing

1.1. Introduction

For a long time, people have been trying to create as much realistic scenes in computer graphics as possible. Finding a way to create photorealistic images, which are indistinguishable from photographs of a real three dimensional scene (see Figure 1.1), has been a big goal for graphics developers for many years. One of the first successful image synthesis methods started with an idea from physics literature. It originated from designing glass lenses by physicists who calculated paths by hand. The paths taken by rays of light starting at the light source were plotted on the paper, they were then passed through the lens and slightly beyond. This process of following the light rays was called *ray tracing* and nowadays it is considered to be one of the most popular rendering methods ([1], [2]).

Figure 1.1: Photograph or computer graphics? André Kirschner

Ray tracing is a technique used for generating images by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects (see Figure 1.2). It is capable of simulating a wide variety of optical effects, such as reflections, refractions, soft shadows or scattering, which provide a high degree of visual realism (for more information see [3]).

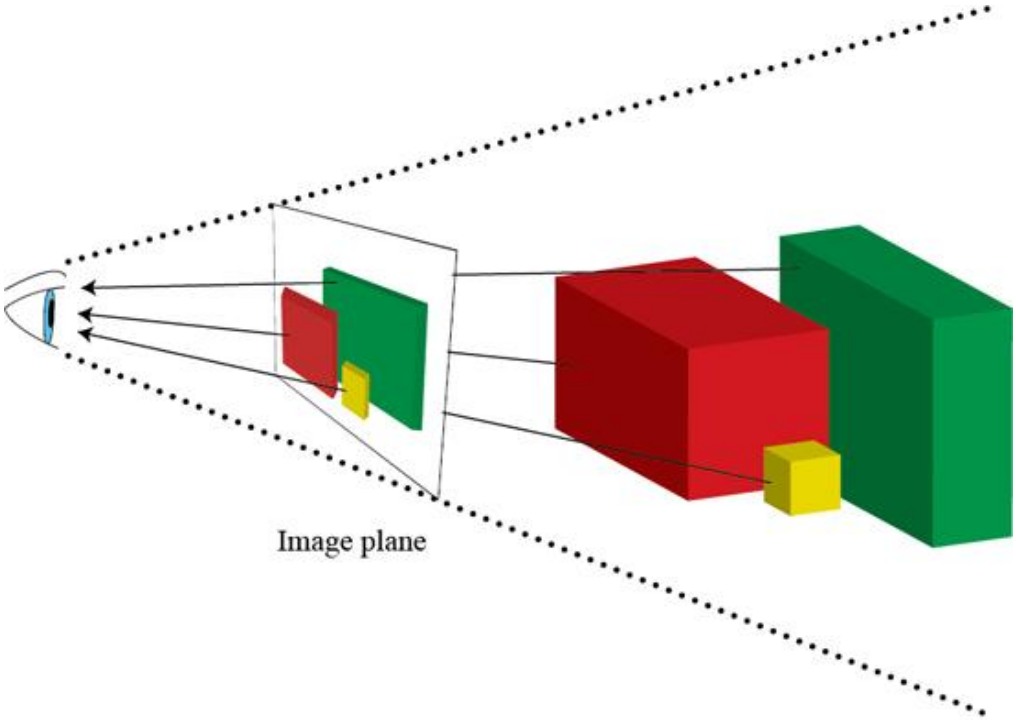


Figure 1.2: Ray tracing

1.2. Algorithm overview

Ray tracing works by tracing a light path and calculating the color of the object visible through it. In the real world, the light source generates possibly millions of light rays that go to slightly different directions. Some of them just pass right out of the scene. Other rays hit objects in the scene, where they reflect, eventually refract, according to the optical character of the objects. Then they bounce around the scene. Some get so dim after couple of bounces, that we can't see them anymore, the rest strike the image plane and pass into the eye. If we decided to create an image by following light rays from the source, it would be very expensive, because too

many rays simply do not play a role in the image. Creation of one dim image might take years due to following pointless rays (see Figure 1.3). The key for computational efficiency is to reverse the problem, by following the rays backward instead of forward - from the eye, to the objects, to the light source (see Figure 1.4). This observation allows us to restrict to the rays that we know will be useful, the ones that really contribute to the image.

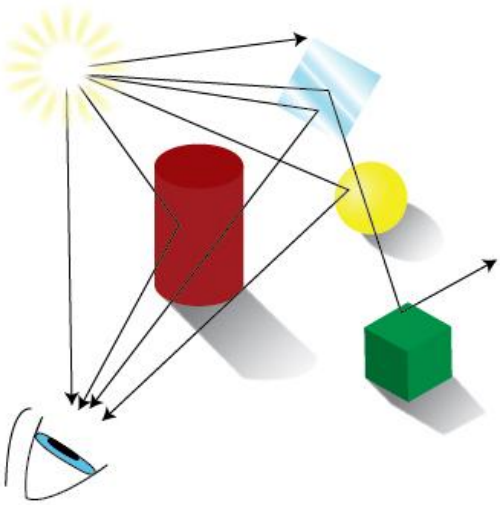


Figure 1.3: Real world

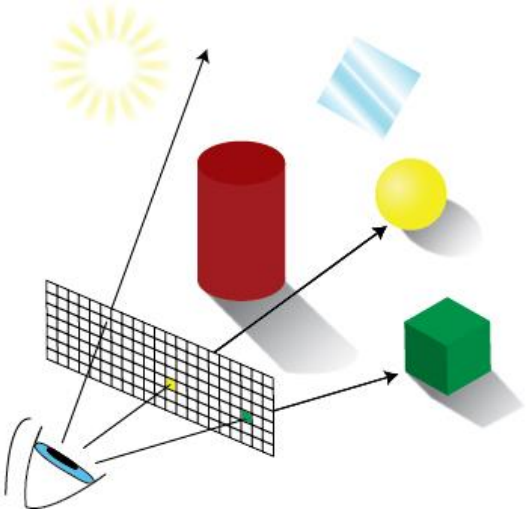


Figure 1.4: Backward ray tracing

We start by asking which rays actually strike the image plane and then pass into the eye. To find out which direction the light ray came from, we have to link the eye with the pixel of the image plane that belongs to the ray, then extend the ray to the scene and look for the first object on the path of the ray. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel.

The most important part of ray tracing is finding the point of intersection between a ray and an object. A ray tracing system includes routines for many types of primitive objects, which the scene can be made up of, such as cuboids, spheres, cylinders, cubes, cones, torus, etc. using constructive solid geometry (CSG). [4]

1.3. Constructive solid geometry

CSG is a procedural modeling technique often used in 3D computer graphics. It allows a modeler to create a complex surface or 3D shapes by using Boolean operators to combine simpler objects (see Figure 1.5, Figure 1.6, Figure 1.7). These simpler objects are called primitives. The shape or surface created by CSG appears visually complex, but it is actually just a cleverly combined or recombined object. It is a solid object with well-defined boundaries and the advantage of such a shape is that it is easy to classify arbitrary points as being either inside or outside the shape (more detailed description can be found in [5], [6]).

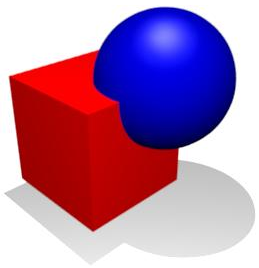


Figure 1.5: Boolean union

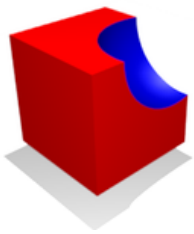


Figure 1.6: Boolean difference

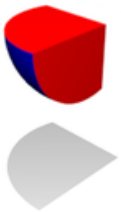


Figure 1.7: Boolean intersection

1.4. Ray tracing recursive implementation

Image plane is actually a raster image (or bitmap), which consist of pixels. How many pixels there are, is set by the resolution of the image. To create the final graphics, the ray tracer has to cast a ray through each pixel (as mentioned in 1.3). The code below describes how the ray tracer traces a ray through one pixel [7].

```
function Trace (P0,P1: Point3D, depth: integer) : RGB;
                {P0 .. beginning of the ray, P1 .. direction,
                                  depth .. number of reflections}
var A, R, T : Point3D          {additional points and vectors}
    B : RGB                                    {resulting color}

begin
  A := Intersection(Scene,P0,P1); {scene and ray intersection}
  if A = 0 then Trace := Background  {ray didn't hit anything}
           else
    begin                                    {ray hit something}
      B := 0;
      for i := 1 to N do          {light sources contributions}
        if Intersection(Scene,A,L[i]-A) = 0
        then B := B + kL*Light(A,L[i]);
      depth := depth + 1;
      if depth <= maxdepth then              {end of recursion}
        begin
          if "A is reflectance" then
            begin
              "compute R"                        {reflected ray}
              B := B + kR*Trace(A,R,depth);
            end;
          if "A is transparent" then
            begin
              "compute T"                        {refracted ray}
              B := B + kT*Trace(A,T,depth);
            end;
        end;
      Trace := B;                  {accumulated release value}
    end;
end;
```

1.5. Limitations of ray tracing

Ray tracing works well by rendering scenes made up of solid objects. These objects have infinitesimally thin boundary surfaces. Their volume is geometrically definable, therefore, every primitive can be exactly described by a formula. After instituting a ray equation into it, we can get at least two intersections – the starting and the ending intersection. Almost any solid can be built from primitives using CSG or by triangle mesh and as an appearance improvement, a texture can be used. But imagine that the object we would like to create is for example phenomena like hair, fur or erosion effect. Such surface is too complex to be created efficiently by using CSG and if we still dare to try it, it would appear smooth and plastic instead of textured and natural. Regarding fire and smoke, they don't even have well defined boundaries at all, so it is hard to find out, if we are inside or outside of them. Therefore, an increased realism is needed. [9]

Chapter 2

Hypertextures

2.1. Introduction

Realism can be achieved by the application of surface details algorithms. One of them is *texture mapping* in one, two or three dimensions. The most diverse of them are two dimensional texture maps. They allow flat two dimensional bitmap images to wrap around the surface of an object (see Figure 2.1). The result suffers from aliasing effect caused by the warping of the texture.

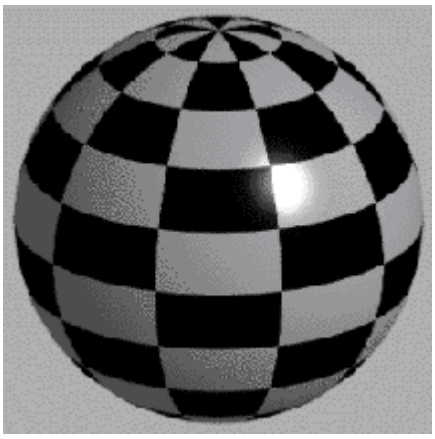


Figure 2.1: Texture mapping in two dimensions (taken from [10])

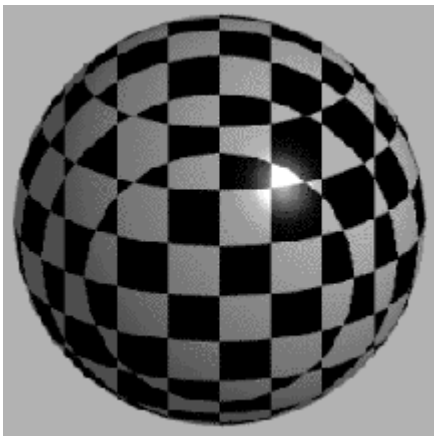


Figure 2.2: Texture mapping in three dimensions (taken from [10])

The need for warping is removed for the three dimensional texture mapping by the fact that the texture is mathematically defined over R^3, thus any point on the surface of the rendered object has a predefined texture value (see Figure 2.2). More information can be found in [10], [11]. Three dimensional texture mapping

offers better results than two dimensional, but there are still plenty of objects (woven materials, fur, etc.) that would look at best awkward when designed by using solid textures. They are limited to the reproduction of objects that has simple surface definitions. Treating object boundaries merely as surfaces has its limitations [9].

2.2. Hypertextures

K. Perlin and E. Hoffert [12] came with an idea that allows production of complex textures through manipulation of surface densities. This technique does not just color the surface of the object with some texture, it changes the surface structure during rendering using a three dimensional texture function. The technique is called *hypertextures*.

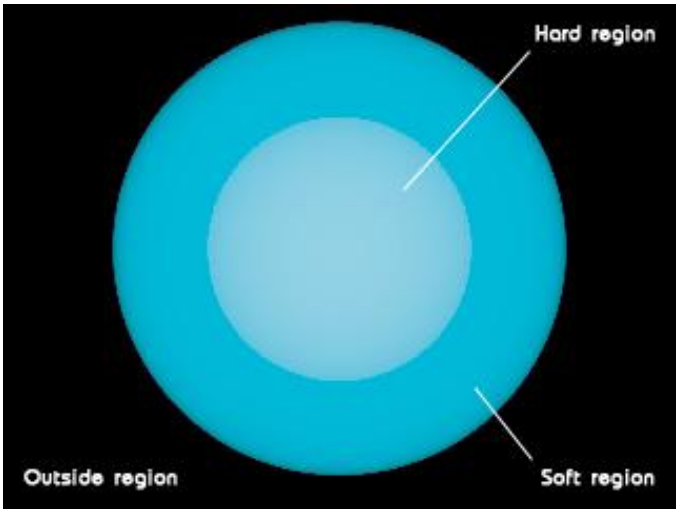


Figure 2.3: Sphere with a soft region

Hypertextures are an extension of solid textures. In comparison to solid textures, hypertexture objects do not have well defined boundaries [13]. Instead they have a density function that describes how the object should behave in the area where it passes between the outside and inside of the object.

Hypertexture objects can be divided into three regions (see Figure 2.3)

- *Soft region*
- *Hard region*
- *Outside region*

In the hard region, the object is completely solid. The soft region represents the part where the object is indeterminate, where the density is variable. In the outside region, an object does not exist.

(The following text was taken from [12]) We have to define:

- *Object Density Function* D(x) with range [0,1] that describes the density of a three dimensional shape for all points x throughout R^3. The soft region consists of all x such that $0 < D(x) < 1$, in a region of nonzero thickness in between.
- *Density Modulation Function* (DMF) f_i, which is used to modulate object's density within its soft region. Each DMF is used to control some aspect of object's spatial characteristics.

Hypertexture is created and formally defined by successive application of DMFs f_i to an object's D(x):

$$H(D(x), x) = f_n (\dots f_2 (f_1 (f_0 (D(x))))) \qquad (1)$$

2.3. Density Modulation Functions

Here are the base level DMFs that higher-order DMFs are built upon, as shown in Eq. 1.

- *Bias* – used to push up or pull down an object's density. It can be defined by a power curve of Eq. 2.

$$bias_b\big(D(x)\big) = D(x)^{\frac{\ln(b)}{\ln(0{,}5)}} \tag{2}$$

- *Gain* – used to make an object's density gradient either flatter or steeper. By increasing or decreasing g in function gain_g, we can increase or decrease the rate at which the midrange of object's soft region goes from 0.0 to 1.0. It can be defined as a combination of two bias curves (Eq. 3)

$$gain_g\big(D(x)\big) = \begin{cases} \frac{bias_{1-g}(2D(x))}{2} & if\ D(x) < 0.5 \\ 1 - \frac{bias_{1-g}(2-2D(x))}{2} & otherwise \end{cases} \tag{3}$$

- *Noise* – used to approximate band-limited white noise, producing a pseudorandom knots in the range (-1,1).

- *Turbulence* – simulates the appearance of turbulent activity by the summation of noise at increasing frequencies. It is defined by Eq. 4:

$$turb(x) = \sum_i abs\left(\frac{noise(2^i x)}{2^i}\right) \tag{4}$$

For more information see [12], [14].

2.4. Ray marching overview

The fact that hypertextures lack well defined surface necessitates usage of algorithm different from ray tracing. We need to use *ray marcher.* The *ray marching* algorithm is used to generate images of hypertextures. A ray *r* is fired from every pixel in

the image plane into the object space. If the ray does not intersect the bounding box of the object, we move to the next pixel for processing. Else if the ray does intersect the bounding box, the entry (μ_0) and exit (μ_1) points for this ray are computed. Ray marching begins at the ray parameter value μ_0, and proceeds at a fixed increment μ, therefore, D(x) is now evaluated at fixed points along the ray according to Eq. 5.

$$x = x_{\mu_0} + k\,\Delta x_{\mu} \tag{5}$$

where

$$k \in N.$$

A simplified model of ray marching algorithm is given in Figure 2.4.

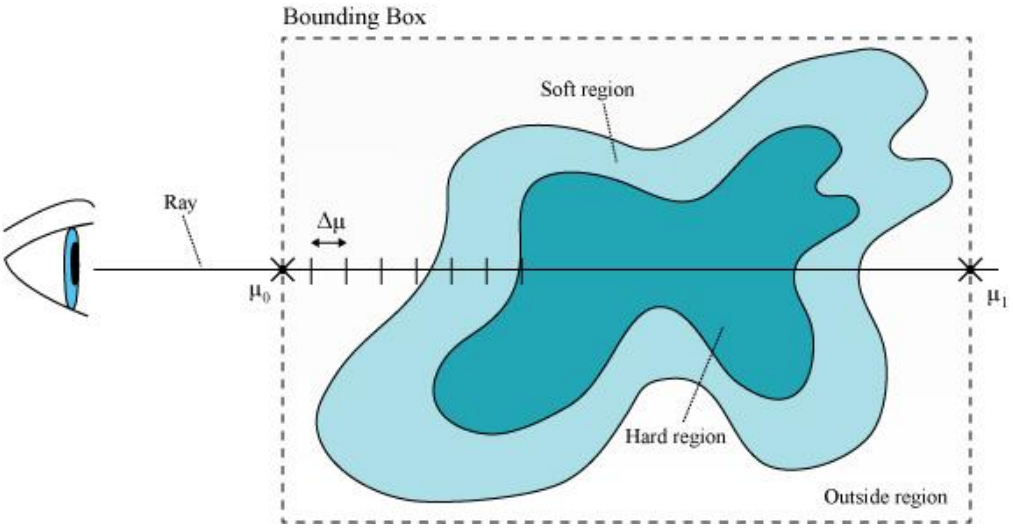


Figure 2.4: Simplified model of ray marching process

If *x* lies within the object's soft region, a DMF(x) is applied and color C_λ (λ = r, g, b) and opacity α are computed for the point of the ray. This process is repeated along the ray, accumulating color and opacity, until one of following condition is reached.

- *Ray termination* - $x_{\mu_0} + k\,\Delta x_{\mu} \geq x_{\mu_1}$
- *Inner surface (hard region) reached* – $D(x) = 1$
- *Opaque surface* - $\alpha \geq 1$

2.5. JaGrLib

According to project specifications, we will carry out this project in JaGrLib. JaGrLib is a library for computer graphics education developed by J. Pelikán and J. Kostlivý ([15],[16]) to primarily help teach of computer graphics at Charles University in Prague. This general modular framework is usable for algorithms and data structures design, testing and tuning. The language used in JaGrLib project is Java, which leads to a high level of flexibility, good portability and comprehensibility. Basic building blocks of the framework (*modules, pieces*) have form of Java classes and the connections between modules are controlled by Java interfaces. These ideas match well with principal rules of object-oriented design and programming.

Each module has one or more *plugs* – exclusive interfaces from inside the module to the rest of the world. Modules can be connected together using *channels*, which can connect one input plug with multiple outputs plugs. (More detailed information about JaGrLib concept can be found in [15].) The whole assembly of modules connected together using channels is called *composition*.

During the formation of the composition, a lot of "manual" routine work is required, for example: choice of suitable module for required interface, replacing functional module with another one, grouping/ungrouping modules to autonomous sub-composition etc. A user-friendly GUI environment to JaGrLib core was developed by Kostlivý [17]. It is called Skel. It is also implemented in Java language with the use of universal graphical interface Swing. Data files are stored in XML format.

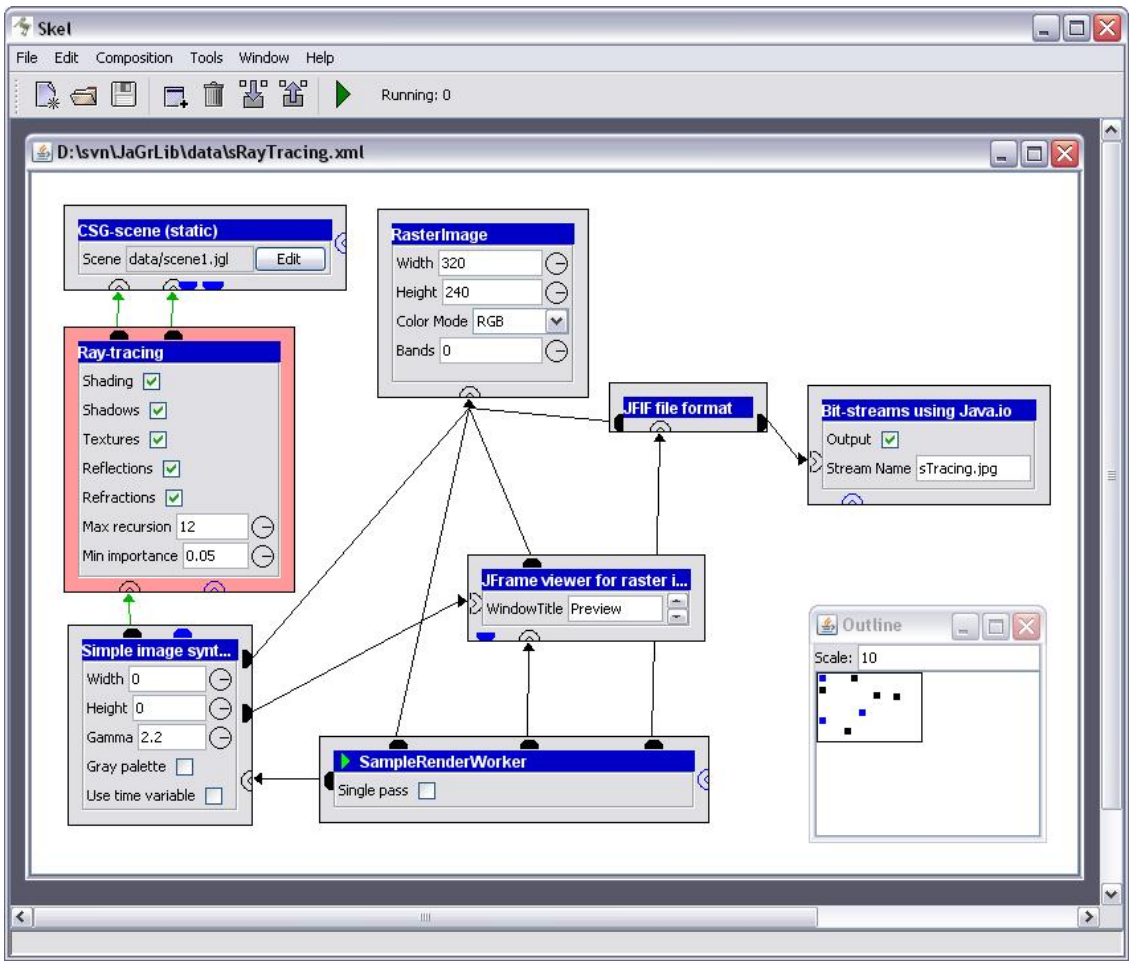


Figure 2.5: Composition of ray tracing

The GUI application looks like a classical MDI frame with number of sub-frames. More compositions can be opened at the same time.

2.6. Ray tracing in JaGrLib

Ray tracing has good support in JaGrLib. [18] The basic components that describe ray tracing in JaGrLib are:

- *Geometric description of the scene (CSG representation)*
- *Light sources and light models*
- *Camera – generator of light rays*
- *Scene definition (scene, light sources, camera)*
- *Image synthesis algorithms (ray tracing)*

The scene is represented using a CSG tree. The base interface is interface SceneNode. An *attributes* concept that is inheritable in the hierarchy is implemented there. The classes implementations of CSG nodes are CSGCommonNode (parent node), CSGNode (inside node including set operations) and CSGLeaf (tree leaf - contains primitive object). Primitive objects implements interface Solid, where the most important functionality is the ability to get intersection with itself (intersection (double[ ], double[ ])), the ability to get required parameters additionally (assertIntersectionAttributes (MicroFacet mf, Object tmp, int attributes)) and the ability to find out if the point lies inside of the object (isInside (double[ ])).

Light sources and light models are defined by interface LightModel and it enables us to count light reflection at the surface of an object. A PhongModel is used as the standard *light model*. Light source implements interface LightSource and provides information e. g. regarding the direction of the light source. Also, an ambient light is implemented.

Camera stands for the generator of light rays or as we mentioned before the eye. It implements interface RayGenerator. It can create a ray in a three-dimensional scene for each point (pixel) of the image plane.

JaGrLib contains tools for XML scripting, *the scene, light sources and camera* for ray tracing is defined using JGL scripts. Scene information is read by the module StaticCSGScene.

The image synthesis algorithm is implemented in two modules. ImageFunction is the interface of the module which returns a color for each pixel of the image plane. Ray tracing is implemented there. ImageSynthesizer creates the raster image RasterGraphics from a common image function.

The modules, which have just been described, can be seen in Figure 2.5.

More detailed information regarding ray tracing in JaGrLib can be found in [18].

Chapter 3

Ray marching implementation

3.1. Introduction

To be able to render hypertextures in JaGrLib, a ray marcher implementation is required. In an implementation of the ray marcher we will make use of the *ray tracing algorithm*, which is extended by the marching algorithm, because the ray marching algorithm differs from the ray tracing algorithm only in marching inside of the soft object. In case the scene is formed not only from soft objects, but also from solids, the ray tracing is used to compute refractions and reflections between all solid objects in the scene. Reflections, refractions and shading are not implemented for hypertextures, we will not be solving it in this project, for it is not the purpose. We will focus on rendering hypertextures only.

3.2. Ray marching

Ray marching algorithm that was described in chapter 2.4, is implemented by the class RayMarching.java. It extends RayTracing, where it adds the method *march*. This method starts only when a ray strikes a soft object, otherwise the shade method continues for solids. The soft object is defined by a property *Boolean isSoftObject* which was newly added to MicroFacet.java. This property is set to true for all soft objects (SoftSphere.java, SoftCube.java).

3.3. Soft objects

Solid objects (defined by interface Solid) that are already implemented in JaGrLib cannot be used to render hypertextures, because there is no soft region that can be modulated by density modulation functions. We must define a new interface Soft, which adds the *density function D* to the parent Solid interface and enables us to implement some specific soft objects. We implemented softSphere and softCube.

SoftSphere centered at *c* of radius *r* and softness *s* can be defined by the density function

$$
\begin{aligned}
&D_{[c,r,s]}(x):\\
&\quad r_1^2 := \left(r - \frac{s}{2}\right)^2\\
&\quad r_0^2 := \left(r + \frac{s}{2}\right)^2\\
&\quad r_x^2 := (x_x - c_x)^2 + \left(x_y - c_y\right)^2 + (x_z - c_z)^2\\
&\quad D := \begin{array}{l} if\ r_x^2 \le r_1^2\ then\ 1.0\ else \\ if\ r_x^2 \ge r_0^2\ then\ 0.0\ else\ \frac{(r_0^2 - r_x^2)}{(r_0^2 - r_1^2)} \end{array}
\end{aligned}
\tag{6}
$$

where r_0 is the outer (D=0) boundary, r_1 is the inner (D=1) boundary and r_x is the radius of the sphere at the point x [12].

Function *isInside* returns true if the point is inside the object. The point *x* is inside the sphere when

$$
|x_x - c_x|^2 + \left|x_y - c_y\right|^2 + |x_z - c_z|^2 \le r^2 \tag{7}
$$

SoftCube centered at *c* of edge length *l* and softness *s* can be defined by the density function

$$
\begin{aligned}
&D_{[c,l,s]}(x):\\
&\quad distance = \max\left(|x_x - c_x|, \left|x_y - c_y\right|, |x_z - c_z|\right)\\
&\quad D := \begin{array}{l} if\ distance \le \frac{l}{2} - s\ then\ 1.0\ else \\ if\ distance \ge \frac{l}{2}\ then\ 0.0\ else\ 1 - \frac{\left(distance\ - \frac{l}{2} + s\right)}{s} \end{array}
\end{aligned}
\tag{8}
$$

where *distance* equals distance from x to the center.

The point *x* lies within the cube when

$$\max\left(|x_x - c_x|, |x_y - c_y|, |x_z - c_z|\right) \leq \frac{l}{2} \tag{9}$$

3.4. DMFs

According to this project specification, we decided to use noise and fractal functions to modulate soft objects' density and thus, we implemented NoisyHypertexture and FractalHypertexture. Density modulation functions implement interface Hypertextures, where the method *compute* evaluates density in point x and returns its color and alpha.

NoisyHypertexture uses noise of frequency *f* and amplitude *1/f *10* to shape the density of the object. The noise is implemented by a class SimpleNoise.java that has already been in JaGrLib. The frequency controls the number of bumps on the surface, the amplitude controls their height. Various frequencies and amplitudes can be set or the missing value will be automatically computed.

FractalHypertexture uses the noise of many different frequencies summed together. The object's shape takes on a characteristic *1/f* fractal appearance.

$$\sum_i \frac{1}{2^i f}\, noise(2^i f x) \tag{10}$$

The same way as it is for NoisyHypertexture, various frequencies and amplitudes can be set. The level of "fractalness" is affected by the number *i* in the sum. The bigger the value *i* is, the more complex the surface looks. With respect to resolution we preset it to 5, which will provide the best results.

3.5. Scenes

There are four prearranged *.jgl files containing different scenes that we have created. JGL script is a XML file with a set of tags corresponding to the basic Java

commands. We can create Java objects (including JaGrLib objects and modules), link them together and return arbitrary set of such instances back to the caller. [20]

The scenes are SoftSphere.jgl, SoftCube.jgl, HypertextureTest.jgl and HypertextureTest2.jgl.

Brief description of the files:

- *SoftSphere* – soft sphere, noisy hypertexture
- *SoftCube* – soft cube, fractal hypertexture
- *HypertextureTest* – soft sphere and soft cube in one scene, both noisy and fractal hypertexture
- *HypertextureTest2* – soft sphere and solid spheres in one scene, noisy hypertexture

The *ATTR_HYPERTEXTURE* attribute was newly added to SceneNode.java to identify hypertexture character of the object. See example code below (A)

```
<object id="leaf1">
  <class>CSGLeaf</class>
  <constructor>
   <ref>softSphere</ref>
  </constructor>
  <method>
   <name>setAttribute</name>
   <member>
    <class>SceneNode</class>
    <name>ATTR_HYPERTEXTURE</name>
   </member>
   <object>
    <class>NoisyHypertexture</class>
    <constructor>
      <val type="I">20</val>
      <val type="D">0.8</val>
    </constructor>
   </object>
  </method>
 </object>                                                    (A)
```

Chapter 4

Results and discussion

All pictures were rendered in OS Microsoft Windows XP Professional, Intel® Core™ Duo CPU T2350, 1,86 GHz, 1,99 GB RAM. Rendering time is just approximative.

Hypertextures were tested on two objects – soft sphere and soft cube. In Figure 4.1 and Figure 4.2 we can see the hard region and the soft region of soft objects. It was rendered using no DMF.

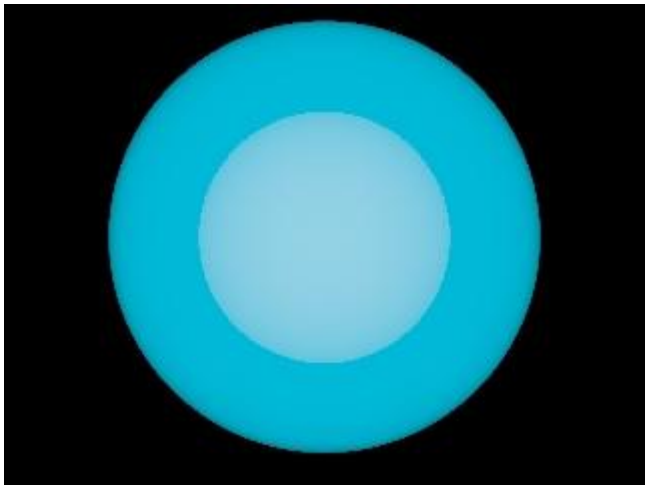


Figure 4.1: Sphere with a soft region

SoftSphere.jgl
No DMF used
Rendering time: 421,5 sec
March size: 0,01
Softness: 0,4

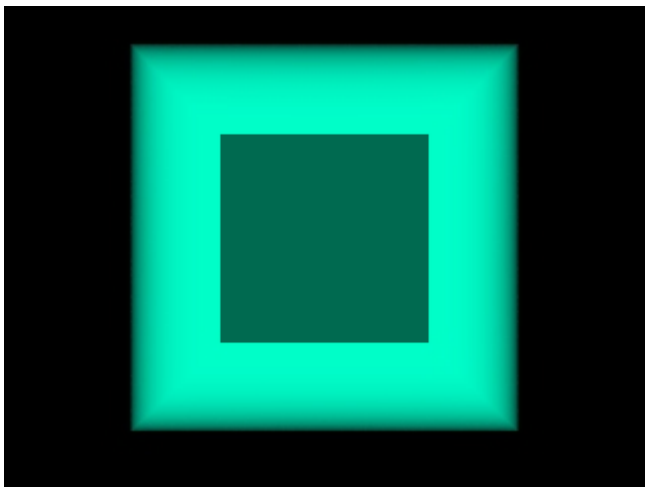


Figure 4.2: Cube with a soft region

SoftCube.jgl
No DMF used
Rendering time: 359,5 sec
March size: 0,01
Softness: 0,4

In Figure 4.3, Figure 4.4, Figure 4.5 and Figure 4.6. we are testing the influence of march size on the final result and the rendering time. Smaller march size adds more details to the object, whereas the bigger march skips them. Logically, the smaller the march size is, the longer the rendering.

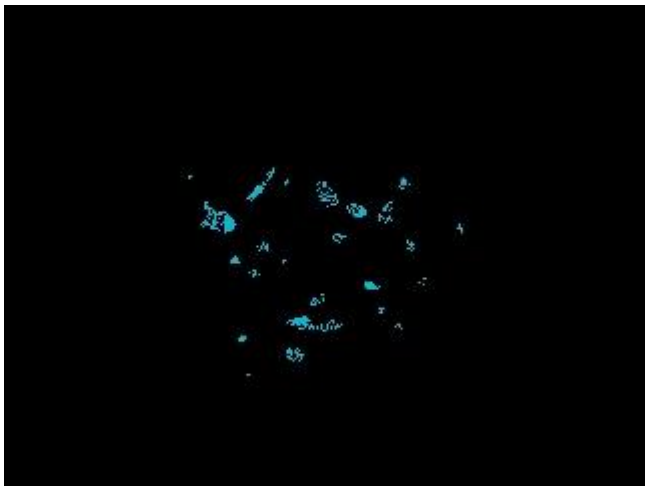


Figure 4.3: Influence of march size (= 1,0)

SoftSphere.jgl
NoisyHypertexture
Rendering time: 3,7 sec
March size: 1,0
Softness: 0,5
Frequency: 20
Amplitude: 0,5

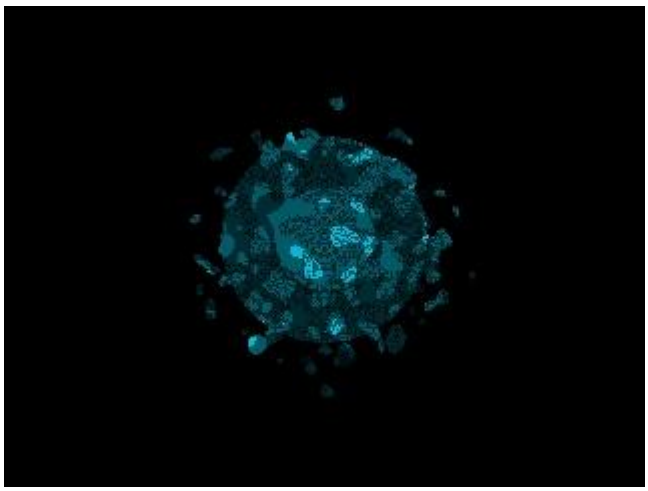


Figure 4.4: Influence of march size (= 0,8)

SoftSphere.jgl
NoisyHypertexture
Rendering time: 9,5 sec
March size: 0,8
Softness: 0,5
Frequency: 20
Amplitude: 0,5

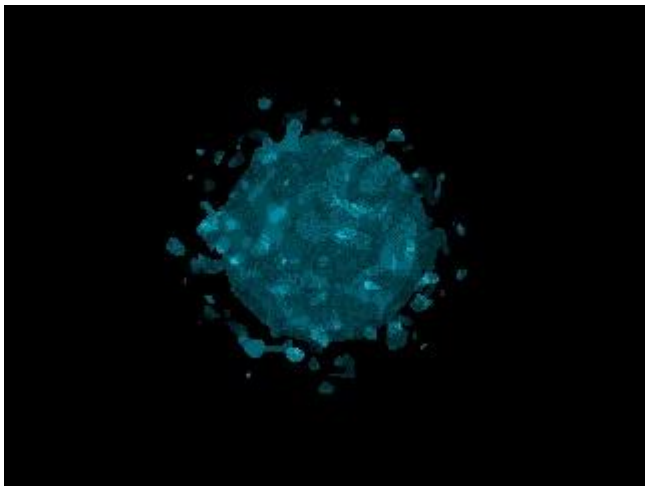


Figure 4.5: Influence of march size (= 0,5)

SoftSphere.jgl
NoisyHypertexture
Rendering time: 14,1 sec
March size: 0,5
Softness: 0,5
Frequency: 20
Amplitude: 0,5

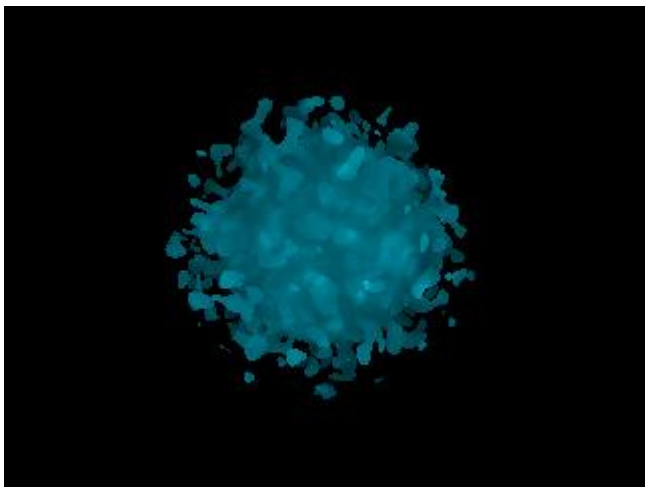


Figure 4.6: Influence of march size (= 0,01)

SoftSphere.jgl
NoisyHypertexture
Rendering time: 549,9 sec
March size: 0,01
Softness: 0,5
Frequency: 20
Amplitude: 0,5

Figure 4.7:, Figure 4.8, Figure 4.9, Figure 4.10 and Figure 4.11 are figures of soft cubes tested with regard to frequency. Amplitude is unchanging, preset to 1.0. We can observe that with increasing frequency the rendering time is decreasing.

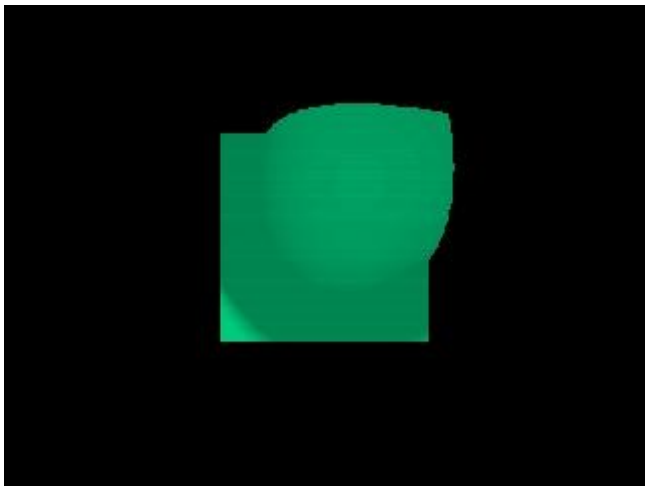


Figure 4.7: Influence of frequency (= 1)

SoftCube.jgl
NoisyHypertexture
Rendering time: 72,5 sec
March size: 0,05
Softness: 0,4
Frequency: 1
Amplitude: 1,0

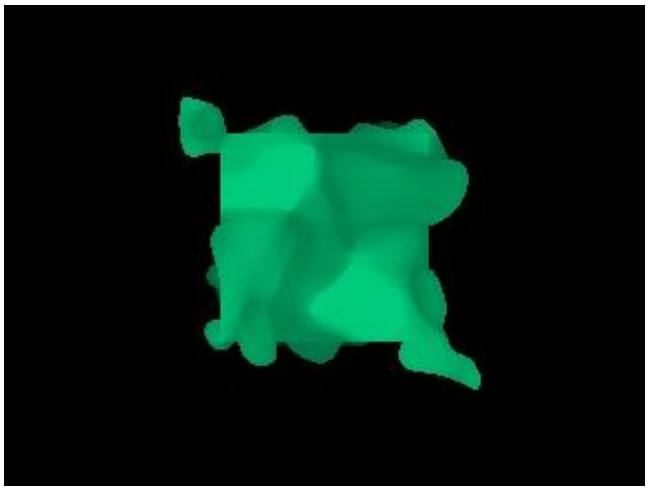


Figure 4.8: Influence of frequency (= 5)

SoftCube.jgl
NoisyHypertexture
Rendering time: 69,5 sec
March size: 0,05
Softness: 0,4
Frequency: 5
Amplitude: 1,0

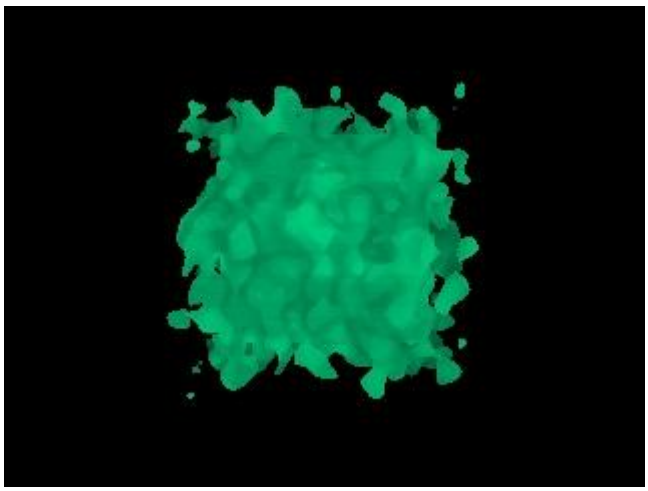


Figure 4.9: Influence of frequency (=12)

SoftCube.jgl
NoisyHypertexture
Rendering time: 55,0 sec
March size: 0,05
Softness: 0,4
Frequency: 12
Amplitude: 1,0

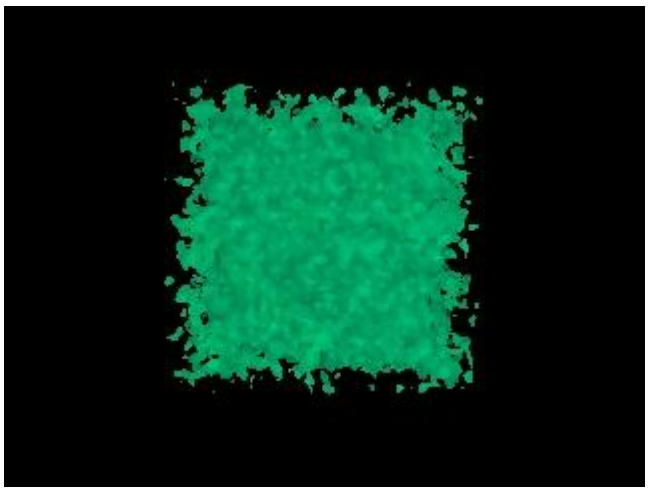


Figure 4.10: Influence of frequency (=25)

SoftCube.jgl
NoisyHypertexture
Rendering time: 48,2 sec
March size: 0,05
Softness: 0,4
Frequency: 25
Amplitude: 1,0

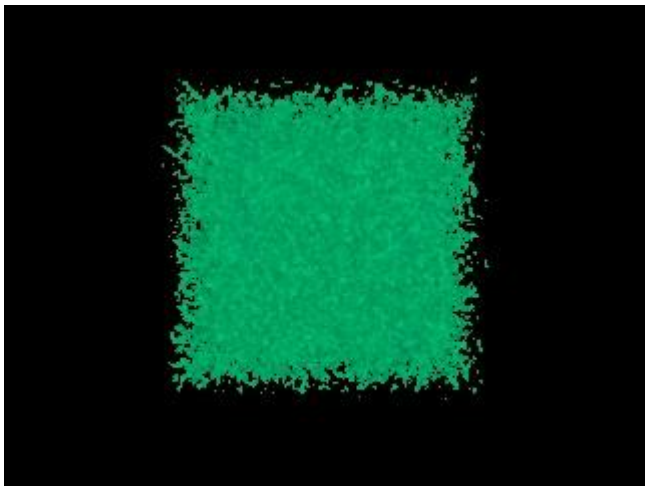


Figure 4.11: Influence of frequency (=50)

SoftCube.jgl
NoisyHypertexture
Rendering time: 45,9 sec
March size: 0,05
Softness: 0,4
Frequency: 50
Amplitude: 1,0

When the amplitude is not set from the constructor of the DMF, it is automatically computed. On Figure 4.12, Figure 4.13, Figure 4.14, Figure 4.15 and Figure 4.16 we can see how hypertexture behave when the amplitude is automatically computed.

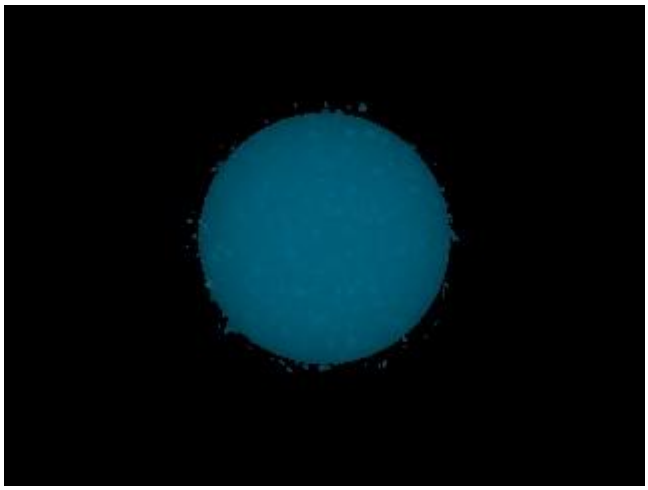


Figure 4.12: Frequency = 40, amplitude = 0,25

SoftSphere.jgl
NoisyHypertexture
Rendering time: 110,3 sec
March size: 0,05
Softness: 0,4
Frequency: 40
Amplitude: 0,25

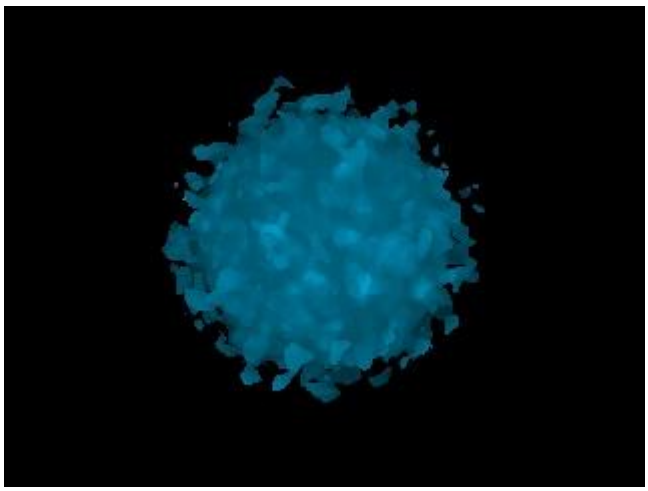


Figure 4.13: Frequency = 20, amplitude = 0,5

SoftSphere.jgl
NoisyHypertexture
Rendering time: 96,5 sec
March size: 0,05
Softness: 0,4
Frequency: 20
Amplitude: 0,5

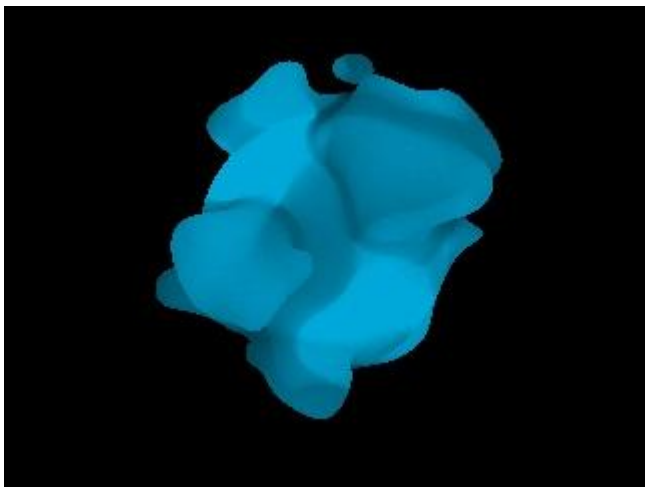


Figure 4.14: Frequency = 5, amplitude = 2,0

SoftSphere.jgl
NoisyHypertexture
Rendering time: 84,5 sec
March size: 0,05
Softness: 0,4
Frequency: 5
Amplitude: 2,0

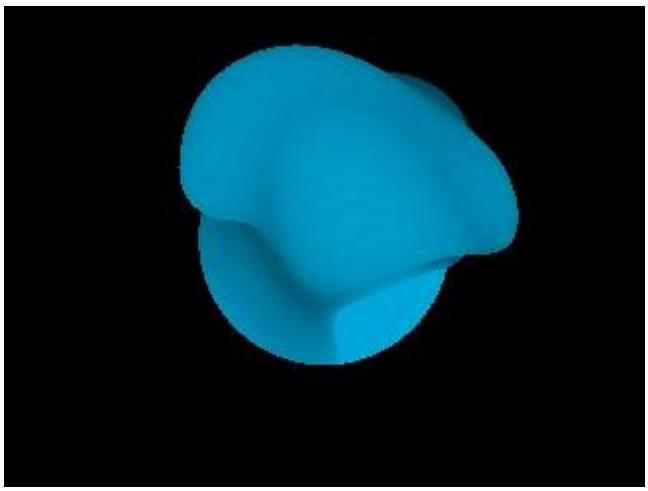


Figure 4.15: Frequency = 2, amplitude = 5,0

SoftSphere.jgl
NoisyHypertexture
Rendering time: 75,3 sec
March size: 0,05
Softness: 0,4
Frequency: 2
Amplitude: 5,0

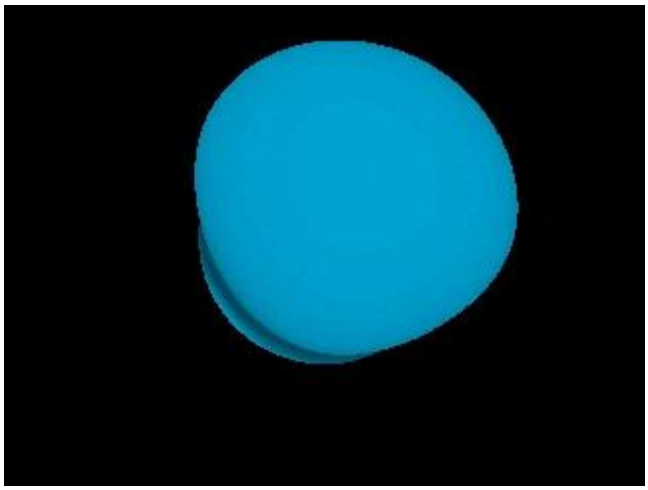


Figure 4.16: Frequency = 1, amplitude = 10,0

SoftSphere.jgl
NoisyHypertexture
Rendering time: 59,0 sec
March size: 0,05
Softness: 0,4
Frequency: 1
Amplitude: 10,0

Soft objects can also differ in softness. The influence of softness on soft objects can be seen in Figure 4.17, Figure 4.18 and Figure 4.19. Both the cube and sphere have the same frequency and the same amplitude. The smaller is the softness, the shorter is the rendering time.

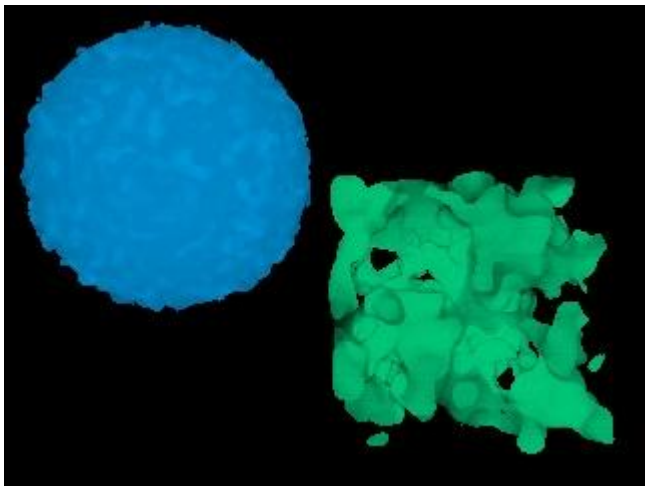


Figure 4.17: Softness influence

HypertextureTest.jgl
Sphere - NoisyHypertexture
Cube - FractalHypertexture
Rendering time: 90,0 sec
March size: 0,05
Sphere softness: 0,1
Sphere frequency: 20
Sphere amplitude: 0.5
Cube softness: 0,9
Cube frequency: 20
Cube amplitude: 4.0
Cube number of steps: 2

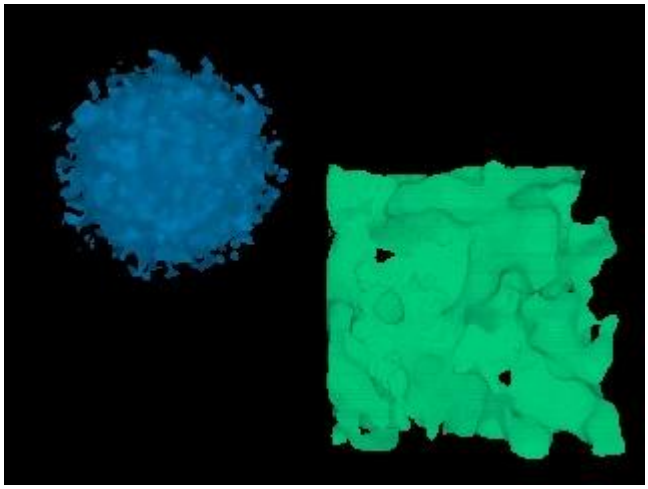


Figure 4.18: Softness influence

HypertextureTest.jgl
Sphere - NoisyHypertexture
Cube - FractalHypertexture
Rendering time: 90,4 sec
March size: 0,05
Sphere softness: 0,5
Sphere frequency: 20
Sphere amplitude: 0.5
Cube softness: 0,5
Cube frequency: 20
Cube amplitude: 4.0
Cube number of steps: 2

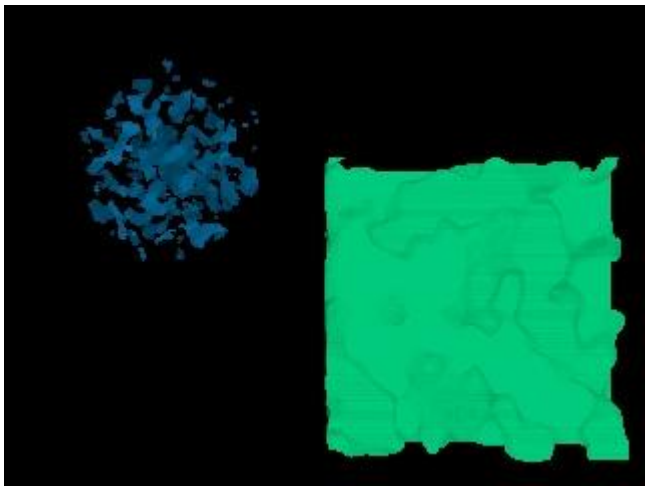


Figure 4.19: Softness influence

HypertextureTest.jgl
Sphere - NoisyHypertexture
Cube - FractalHypertexture
Rendering time: 71,6 sec
March size: 0,05
Sphere softness: 0,9
Sphere frequency: 20
Sphere amplitude: 0.5
Cube softness: 0,1
Cube frequency: 20
Cube amplitude: 4.0
Cube number of steps: 2

Figure 4.20, Figure 4.21, Figure 4.22, Figure 4.23 and Figure 4.24 show the influence of the number of steps in FractalHypertexture sum. We can see that the complexity and "fractalness" increases with the increasing number of steps. The bigger the value is, the more expensive it is to render the image.

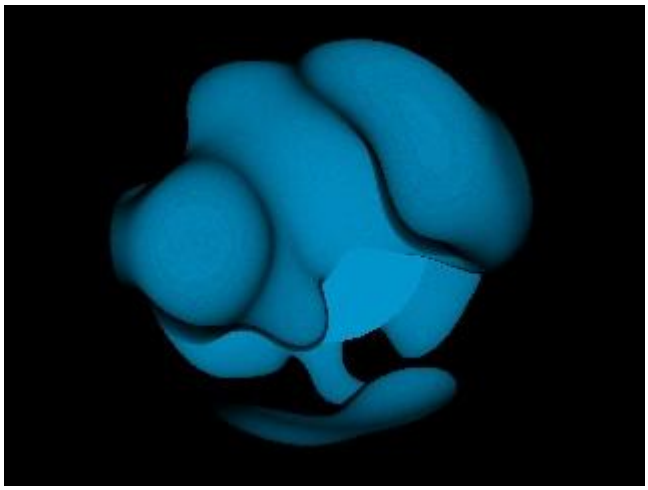


Figure 4.20: Number of steps = 1

SoftSphere.jgl
FractalHypertexture
Rendering time: 326,2 sec
March size: 0,01
Softness: 0,5
Frequency: 3
Amplitude: 10,0
Number of steps: 1

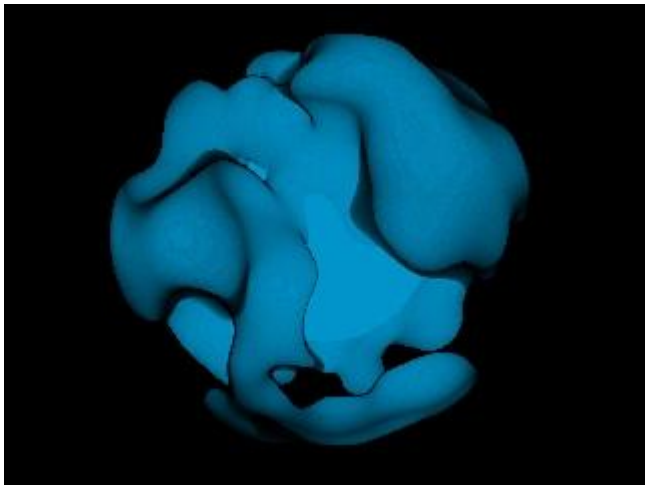


Figure 4.21: Number of steps = 2

SoftSphere.jgl
FractalHypertexture
Rendering time: 528,0 sec
March size: 0,01
Softness: 0,5
Frequency: 3
Amplitude: 10,0
Number of steps: 2

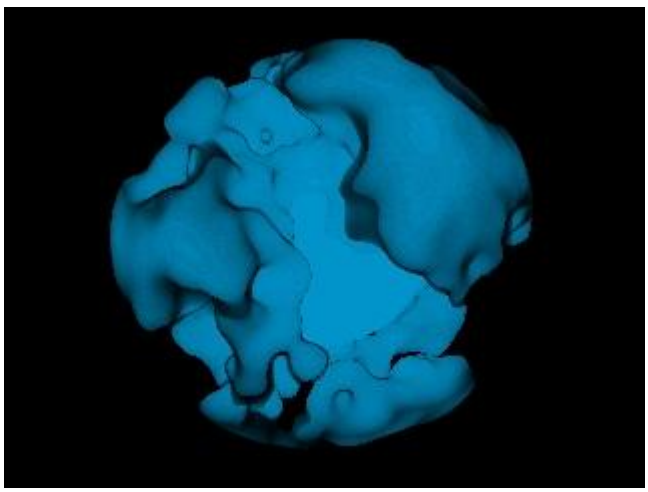


Figure 4.22: Number of steps = 3

SoftSphere.jgl
FractalHypertexture
Rendering time: 686,9 sec
March size: 0,01
Softness: 0,5
Frequency: 3
Amplitude: 10,0
Number of steps: 3

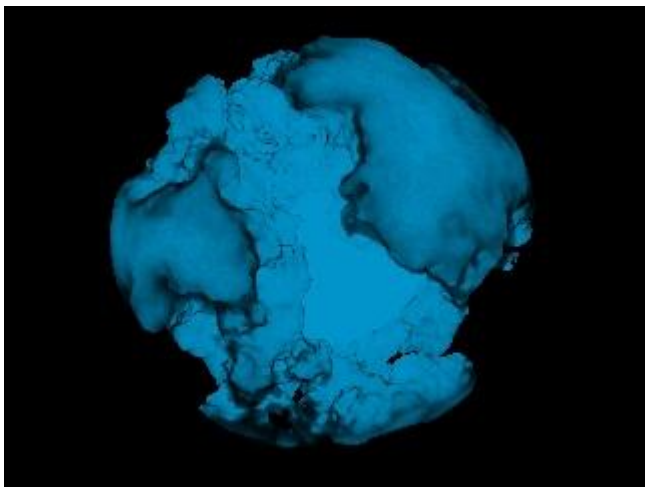


Figure 4.23: Number of steps = 5

SoftSphere.jgl
FractalHypertexture
Rendering time: 1106,0 sec
March size: 0,01
Softness: 0,5
Frequency: 3
Amplitude: 10,0
Number of steps: 5

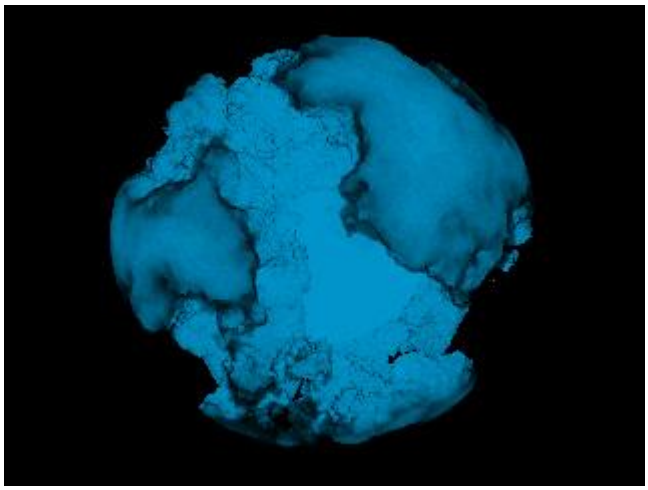


Figure 4.24: Number of steps = 20

SoftSphere.jgl
FractalHypertexture
Rendering time: 4126,6 sec
March size: 0,01
Softness: 0,5
Frequency: 3
Amplitude: 10,0
Number of steps: 20

The effect of different FractalHypertexture frequencies can be seen in Figure 4.25, Figure 4.26, Figure 4.27 and Figure 4.28. Just as NoisyHypertexture, the time increases with increasing frequency.

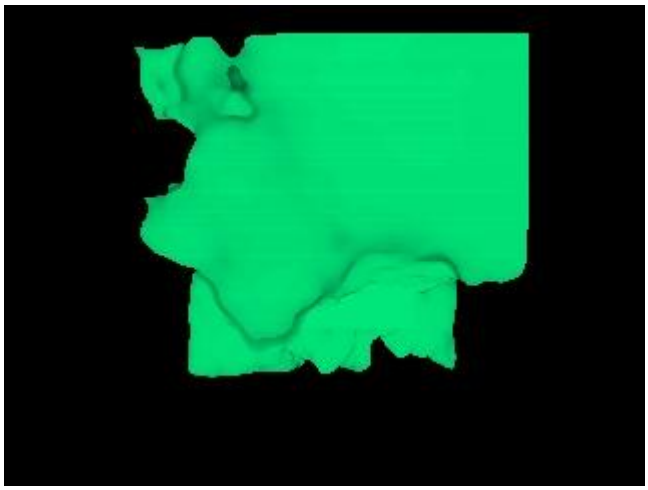


Figure 4.25: FractalHyp. frequency = 1

SoftCube.jgl
FractalHypertexture
Rendering time: 1132,5 sec
March size: 0,01
Softness: 0,5
Frequency: 1
Amplitude: 8,0
Number of steps: 5

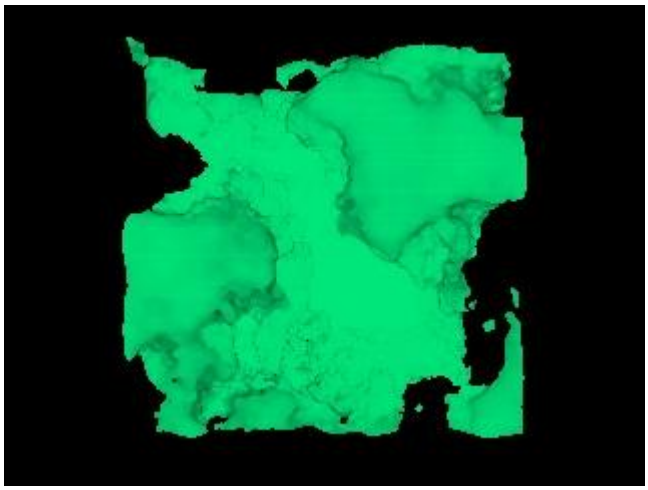


Figure 4.26: FractalHyp. frequency = 3

SoftCube.jgl
FractalHypertexture
Rendering time: 884,9 sec
March size: 0,01
Softness: 0,5
Frequency: 3
Amplitude: 8,0
Number of steps: 5

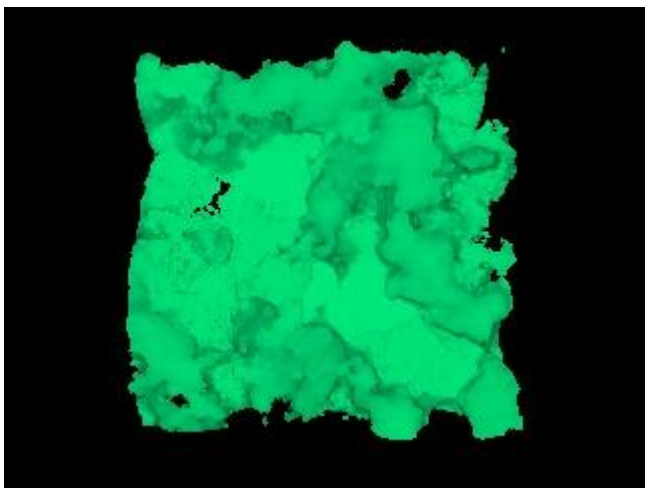


Figure 4.27: FractalHyp. frequency = 5

SoftSphere.jgl
FractalHypertexture
Rendering time: 703,1 sec
March size: 0,01
Softness: 0,5
Frequency: 5
Amplitude: 8,0
Number of steps: 5

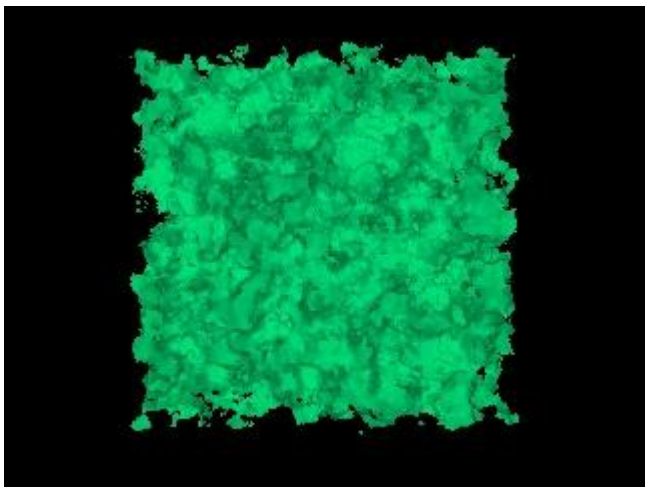


Figure 4.28: FractalHyp. frequency = 15

SoftCube.jgl
FractalHypertexture
Rendering time: 595,9 sec
March size: 0,01
Softness: 0,5
Frequency: 15
Amplitude: 8,0
Number of steps: 5

Finally, on the following image (Figure 4.29), we can see a test on how soft objects interact with solid objects. This scene was modified from *phongTest.jgl*, where a sphere with texture was replaced with a sphere with hypertexture. Seeing that ray tracing algorithm was just extended to the *march* method (to get ability to depicture hypertextures), behaving of the rendering of solids and their interactions between themselves was not changed. We can see reflections and refractions on the surfaces of solid objects and how the solid spheres are "sunk" in a shadow according to the position of the light source. Neither reflections and refractions computation nor shading was implemented for hypertextures. Future extension is recommended.

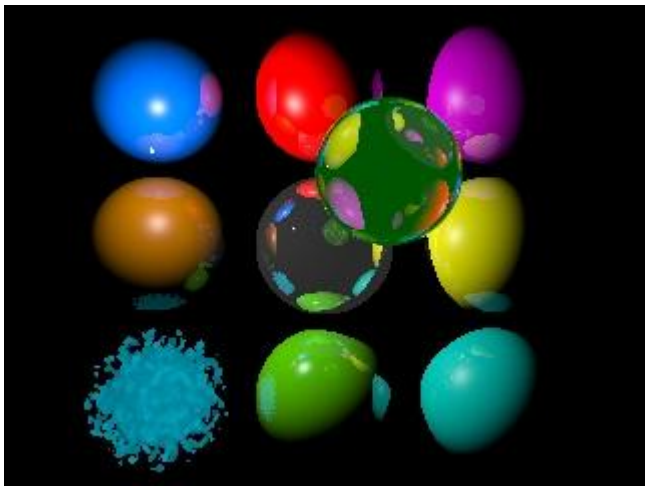


Figure 4.29: Soft object and solid objects

HypertextureTest2.jgl
NoisyHypertexture
Rendering time: 63,3 sec
March size: 0,01
Soft sphere softness: 0,5
Frequency: 20

Amplitude: 0.5

4.1. Time complexity

Rendering hypertextures is quite expensive. The time complexity of the algorithm is $O(n^3)$ with respect to the image resolution.

Final rendering time can be affected by:

- *March size* – the smaller the march size, the longer the image renders, but it's not true for all values, as by decreasing the march size will cause niceness to increase. It was found that hypertextures rendered with march size values e.g. 0,01 and 0,001 do not significantly differ, but the rendering time is ~10 times longer for 0,001. Smaller march size makes sense for bigger resolution.
- *Softness* – the softer the object is, the slower the rendering is.
- *Frequency, amplitude* – it was observed that higher frequency (amplitude) accelerates the time of rendering.
- *Number of steps i (fractalHypertexture)* – the time complexity grows with increasing *i*. Same as for the march size, bigger values make sense for bigger resolution.

Rendering acceleration could be a good tip for further development.

Chapter 5

User guide

5.1. JaGrLib and Skel

JaGrLib is required to be able to work with this project. Both JaGrLib and Skel are part of one SVN repository. The URL address of the repository is *svn://cgg.mff.cuni.cz/JaGrLib/trunk* and it can be downloaded using SVN client. JaGrLib uses Java, oldest version 1.6 (J2SE version 6). How to setup JaGrLib in Eclipse or NetBeans IDE, how to work with it and much more detailed information regarding this can be found in [19].

5.2. Starting application

Start Skel by compiling the JaGrLib project. Press *Load Composition* button and find *RayMarching.xml* in */JaGrLib/data* folder. Ray marching composition will open. In comparison to Ray Tracing module, Ray Marching module contains one new property, which is *march size*. We can change it to get different results. Smaller march size gives a more detailed resolution.

5.3. Changing the scene

There are six prearranged scenes that can be ray march tested. They are

- SoftSphere.jgl
- SoftCube.jgl
- HypertextureTest.jgl
- HypertextureTest2.jgl

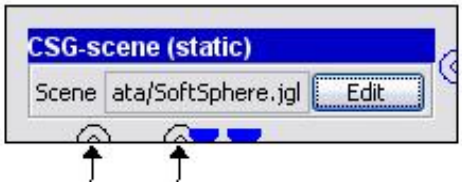


Figure 5.1: Scene module in Skel

The predefined scene is SoftSphere.jgl as it contains only one sphere, where the DMF is preset to NoisyHypertexture. SoftCube.jgl is similar – contains one cube, but the default DMF is FractalHypertexture. HypertextureTest.jgl consists of noisy soft sphere and fractal soft cube defined by different parameters. HypertextureTest2.jgl contains both soft and solid objects. The following chapter 5.4. will explain how to change the parameters.

To change the scene, press the Edit button in CSG-scene (static) module (see Figure 5.1) and select a different scene. All scenes can be found in the */JaGrLib/data* folder.

5.4. Changing properties of soft objects

The appearance of soft objects can be modified in few ways. The first way is to change the thickness of the soft region that is shaped by the density modulation functions (see Figure 5.2). *Softness* can be set in the *.jgl file of the scene in a soft object's constructor. The type of softness value is double. See example code (B) below.

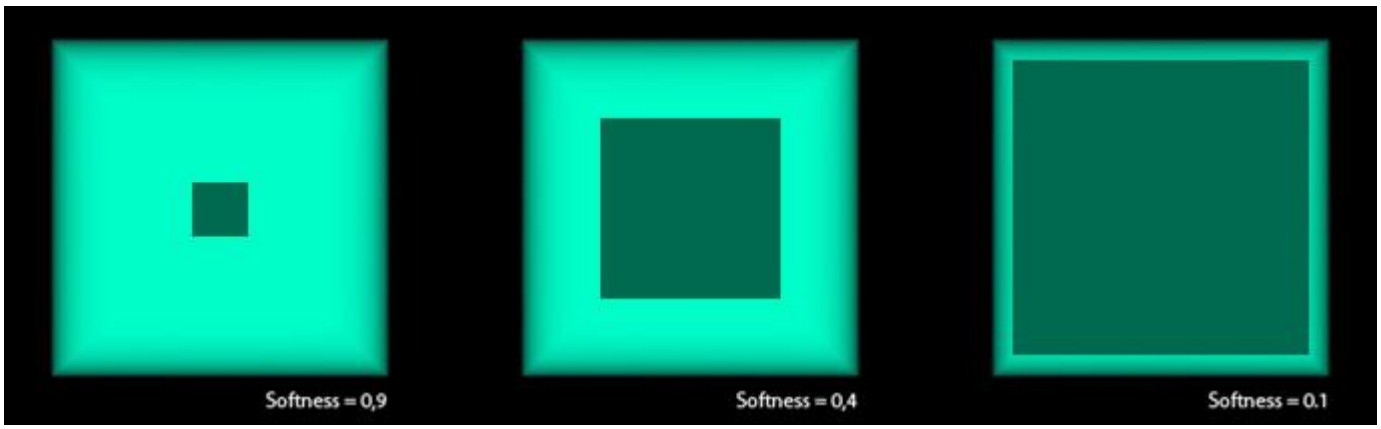


Figure 5.2: Different thickness of the soft region of a cube

```
<object id="softCube">
  <class>SoftCube</class>
  <constructor>
    <val type="D">0.4</val>
  </constructor>
 </object>                                                    (B)
```

The second way is to change the density modulation function. We have implemented two DMFs - NoisyHypertexture and FractalHypertexture. DMF can be also set in *.jgl file containing the scene.

NoisyHypertexture: The user can affect the look of the hypertexture by the setting noise frequency and amplitude. Frequency type is integer, amplitude type is double. Both or one parameter can be set, while the second is automatically computed. If there are no values in the constructor, preset values will be used. See example code (C).

```
<object>
 <class>NoisyHypertexture</class>
 <constructor>
   <val type="I">30</val>
   <val type="D">1.2</val>
 </constructor>
</object>                                                    (C)
```

FractalHypertexture: Fractal DMF adds the typical fractal 1/f look to the object. Same as the NoisyHypertexture, various frequencies and amplitudes should be set to change the appearance. The third parameter that can be set is sum index (integer).

The higher the value of sum index, the more “fractal” the object looks. See example code below (D).

```
<object>
    <class>FractalHypertexture</class>
    <constructor>
      <val type="I">5</val>
      <val type="D">8.0</val>
      <val type="I">2</val>
    </constructor>
</object>                                                    (D)
```

Figure 5.3: Bit-streams module

5.5. Result

The creation of the final result image is provided by the Bit-Streams module (see Figure 5.3). The name of the final image can be changed here. Result images can be found in the */JaGrLib/* folder.

Chapter 6

Conclusion

According to our task, we have implemented hypertextures into the graphical library JaGrLib that will enable us to obtain visually realistic representations of objects with very complex surface definition. This method contrasts with previous techniques that are unable to generate such objects or it is very difficult for them, because they manipulate the matter only on the surface.

Noise and fractal functions were used to manipulate the objects density and were tested on soft objects as a sphere and cube.

Rendering is accomplished by the ray marching algorithm that evaluates and accumulates opacity along each ray throughout volumetric region. The time complexity of the algorithm is $O(n^3)$ with respect to the image resolution.

Thus, we can consider the overall result a success. All expectations set at the beginning have been met. Nevertheless, there will always be space for improvement. Further development includes the extensions of reflection and refraction effects, new soft objects, more density modulation functions and also an acceleration of the rendering.

Bibliography

[1] Glassner, A.S.: An Introduction to Ray Tracing, Academic Press, ISBN 0-12-286160-4, 1989

[2] Friend, Ch.: Ray Tracing - Part 1, www.inaudible.co.uk/temp/RayTracing.ppt

[3] Ray Tracing (Graphics) – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

[4] Kuchkuda, R.: An Introduction to Ray Tracing, University of North Caroline

[5] Constructive Solid Geometry – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Constructive_solid_geometry

[6] Leadwerks Corporation, What is Constructive Solid Geometry, http://www.leadwerks.com/files/csg.pdf

[7] Pelikán, J.: Rekurzivní sledování paprsku, http://cgg.mff.cuni.cz/~pepca/lectures/pdf/raytracing.pdf

[8] Triangle Mesh - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Triangle_mesh

[9] Satherley, R., Jones, M.: Non-Geometrically Definable Volume Data, http://cs.swansea.ac.uk/~csmark/PDFS/vg99.pdf

[10] Okino, Computer Graphics, 2D and 3D Texture Mapping Support, http://www.okino.com/new/toolkit/1-11.htm

[11] Procedural texture – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Procedural_texture

[12] Perlin, K., Hoffert, E., Hypertexture, Computer Graphics, Volume 23 (3)

[13] Byrd, Ch.: Hypertextures http://web.cs.wpi.edu/~matt/courses/cs563/talks/cbyrd/pres2.html

[14] Satherley, R., Jones, M.: Hypertexturing Complex Volume Objects

[15] Pelikán, J., Kostlivý, J.: JaGrLib: library for computer graphics education, http://cgg.mff.cuni.cz/~pepca/papers/wscg2004jagrlib.pdf

[16] Pelikán, J., Kostlivý, J.: JaGrLib: library for computer graphics education, http://cgg.mff.cuni.cz/JaGrLib/

[17] Kostlivý, J.: GUI for the JaGrLib library, Bc. thesis, Charles university, 2003

[18] Ray Tracing v JaGrLibu http://cgg.mff.cuni.cz/JaGrLib/doc/rt.html

[19] JaGrLib: Kuchařka

http://cgg.mff.cuni.cz/JaGrLib/doc/howto.html

[20] Scripting in JaGrLib http://cgg.mff.cuni.cz/JaGrLib/doc/scripting.html

Appendix

The accompanying DVD is organized as follows:

- /Documents – this document in the portable document format
- /Pictures – rendered hypertextures
- /svn – actual repository snapshot with source code of newly implemented ray marching method
- /svn/JaGrLib/data – new *.jgl scenes